UNIVERSITÄT ZU LÜBECK
INSTITUT FÜR
THEORETISCHE INFORMATIK

Implementation and Analysis of Geometric Algorithms for Translucency Sorting in Minecraft

Masterarbeit

douira

22. April, 2024

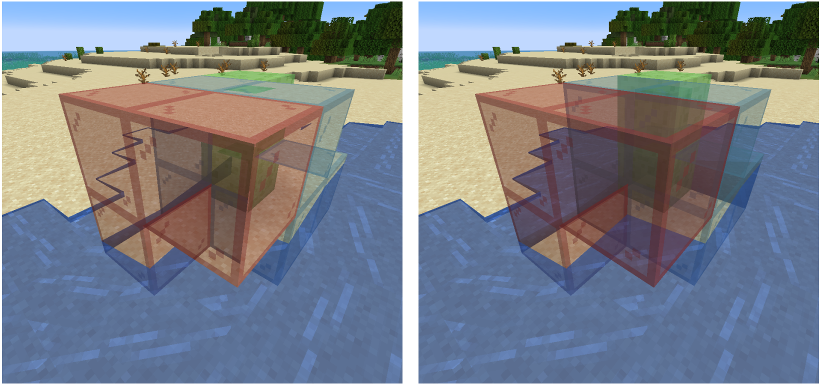
Institut für Theoretische Informatik
Universität zu Lübeck

Contents

1. Motivation and Introduction
2. Rendering and Simple Translucency Sorting
3. Sorting with Graphs
4. Sorting with Partition Trees
5. Unaligned Partitioning
6. Conclusion

Motivation and Introduction

Translucent Rendering in Sodium



(a) Incorrect rendering

(b) Correct rendering

Figure 1: A Minecraft scene with translucent blocks showcasing the problem.

→ This work adds correct translucent rendering to Sodium

→ Minecraft implements it too, but slower and with errors

Translucency in General

- **Translucency** is common in computer graphics
- Translucency $\neq$ Transparency
- Minecraft/Sodium use **ordered rendering** of quads with alpha-blending
- Alternative approach: **Order-independent transparency**
  - Visual inaccuracy
  - Ground-truth variants slower
  - Incompatibility with targeted platforms

Contributions of the Thesis

- Implementation of indexed translucent rendering in Sodium
- Improvements to distance sorting
- Special case handling to simplify sorting where possible
- Topological visibility graph sorting
- Plane-based sort triggering
- Partition tree sorting
- Analysis of unaligned partitioning

Rendering and Simple Translucency Sorting

Rendering Quads

- Blocks are rendered as multiple one-sided **quads**
- Each 16 × 16 × 16 block section is meshed after any change

→ **Alpha blending**: interpolation with alpha factor, non-associative

→ Back-to-front ordering required for correct image

- The GPU renders quads in the order of the **index buffer**

The Translucency Sorting Problem

Problem (Quad-based Translucency Sorting)

- *Given is a set of* **one-sided** *quads Q, each consisting of four vertexes* $q = (v_1, v_2, v_3, v_4) \in (\mathbb{R}^3)^4$ *with* $V_q = \{v_1, v_2, v_3, v_4\}$*. Each quad's vertexes lie in the same plane* $P_q = (n_q, d_q)$ *with the unit* **normal vector** $n_q \in \mathbb{R}^3$ *and* **distance** *from the origin* $d_q \in \mathbb{R}$*.*
- *Quad-based translucency sorting entails finding a* **permutation** *S of the quads in Q such that quads with greater depth are rendered before those with lower depth.*
- *A quad q may be* **omitted** *from S if it is facing away from the camera's position* $c \in \mathbb{R}^3$ *and thus is not visible.*

Unsortable Instances

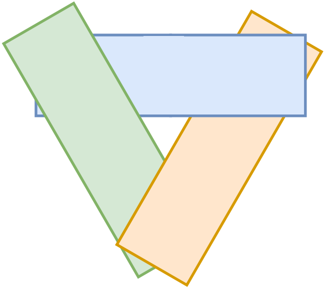


Figure 2: Cyclically overlapping geometry is unsortable.

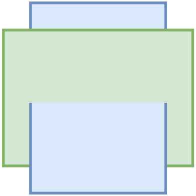


Figure 3: Intersecting geometry is unsortable.

→ Unsortable geometry requires fragmenting quads or per-pixel sorting

Simple Special Cases

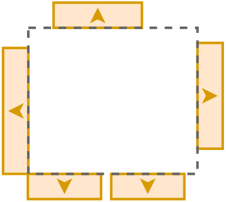


Figure 4: No sorting required if aligned to the bounding box.

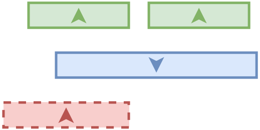


Figure 5: No sorting required if opposite facing and only one dot product value.

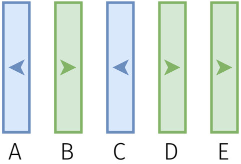


Figure 6: Quads with **two opposing orientations** can be sorted by dot product. (SNR Sorting)

Sort Types

- **Static**: One sort order is correct for all view points
- **Dynamic**: Updated as camera moves and re-sorting is triggered
- Data gathered during meshing determines sort type
- Sort types:
  - None
  - SNR (Static Normal-Relative)
  - Static Graph Sort
  - Dynamic Graph Sort/Partition Tree

Distance Sorting

- Sorting by distance from the camera to the center of each quad

→ Correct in many cases, but not universally, even with frequent sorting

- **Invalidation is unspecific** and is only estimated
- Used as a **fallback** when all other methods fail

(a) Aligned geometry

(b) Unaligned geometry

Figure 7: Center distance sorting is not correct in some situations.

Sorting with Graphs

Topological Sorting of the Visibility Graph

- Sorting based on potential **visual overlap**

→ One correct sort order for a set of view points: **Camera polytope** C

- Sorting the **visibility graph** G_C topologically yields a sort order

Definition (Accurate Visibility Condition)

- A quad p is **visible through another quad** q, referred to as q *seeing* p, if there is a view ray from a position $s \in C$ to a point $t \in S_p$ that intersects with a point in S_q. S_x is the surface of a quad $x \in Q$.
- Additionally, both quads must be facing the camera to be visible at all with $(s - t) \cdot n_p > 0$ and $(s - t) \cdot n_q > 0$.

Dynamic Sorting With the Camera Polytope

→ Camera polytope is constrained by **planes of invisible quads**

- Camera exits the camera polytope when a quad becomes visible
- Triggering is implemented with a combination of interval trees and sorted lists of dot products

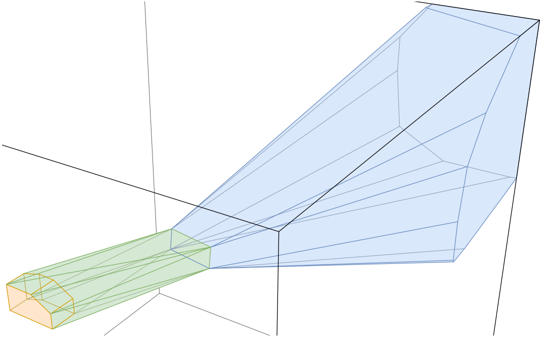


Figure 8: A quad q, the camera polytope C in orange, the visible space T_{Cq} in blue (rendered before q), and the intermediary space in green (rendered after q).

Complexity of Visibility Graph Sorting

- The camera polytope has constant complexity with **quantized normals**
- Evaluating the visibility condition takes constant time
- Visibility is **not transitive**
- Topological sorting may need to process $O(|Q|^2)$ edges

→ Finding a visible neighbor efficiently is hard

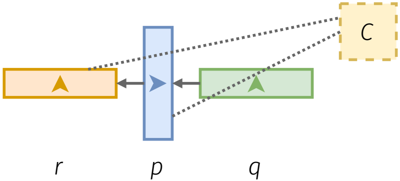


Figure 9: The arrows indicate the visibility relation, and view rays are represented by dotted lines. The relation is not transitive since q cannot see r.

The Implementation of Graph Sorting and its Limits

- Static sorting for limited size instances with **reduced visibility condition**
  - Avoids solving a linear program for the accurate visibility condition
  - Assumes visibility from anywhere: $C = \mathbb{R}^3$
  - Permits testing visibility with just axis-aligned bounding boxes
- For non-static sorting, **separators** alleviate some cycles
- Finding unaligned separators is infeasible

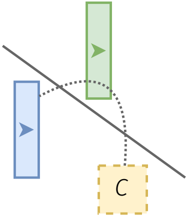


Figure 10: The separator proves that the blue quad is not visible through the green quad from any point in C. A curved view ray is impossible.

Sorting with Partition Trees

Valid and Useful Partitions

Definition (Valid Partition)

- A partition $R = (g, d)$ of $\mathbb{R}^3$ is a plane described by normal $g \in \mathbb{R}^3$ and distance $d \in \mathbb{R}$.
- A partition is valid, also called a **free cut**, if no quads are intersected by the partition plane.

Definition (Useful Partition)
A partition R is useful if it partitions Q into at least two non-empty subsets.

Partition Trees

- A tree arises from **recursive multi-partitioning**

→ Indexes are written for partitions in back-to-front order from the camera perspective

- It always generates a **sort order in linear time**
- Sorting is triggered when the camera crosses a partition plane

→ Construction in $O(n^2 \log n)$, in $O(n \log^2 n)$ if balanced

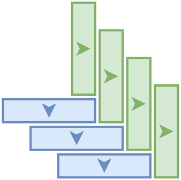


Figure 11: The tree degenerates to a list in the worst case.

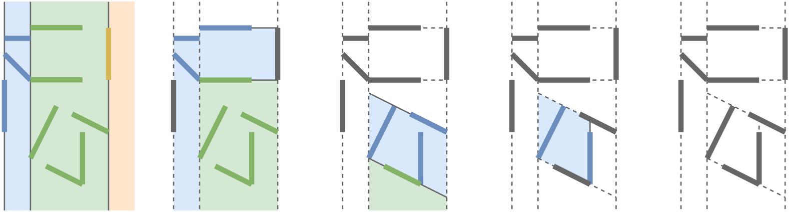


Figure 12: An example of recursive multi-partitioning on 2D lines.

Multi-Partitioning With Known Orientation

1. Projects the quads along a known orientation
2. Sorts the interval endpoints
3. Scans the list for gaps that permit partitions

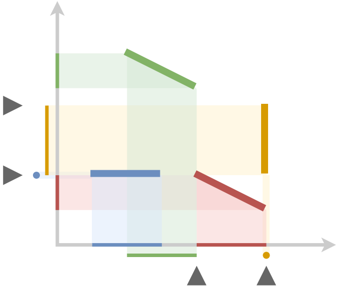


Figure 13: Projection along two axes yields two gaps for each one.

→ The orientation is known: Axis-aligned partitioning only

- Generates as many partition planes as possible
- Quads can be on the partition plane

Unpartitionable Instances

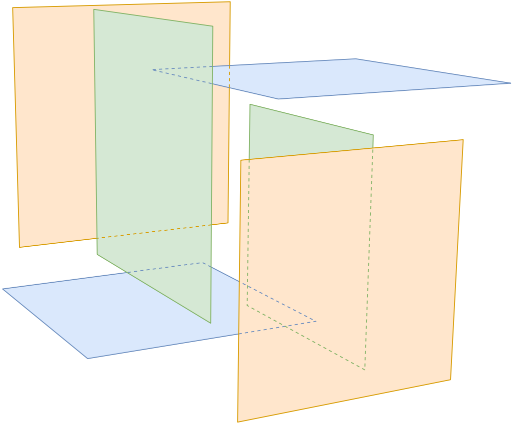


Figure 14: These aligned quads are not partitionable, even when certain quads are removed.

Figure 15: These unaligned quads are not partitionable.

Extensions and Special Cases

- Detecting special cases during partitioning allows for fewer nodes and partition planes
- **Partial tree updates** improve partitioning speed for large sections
- Index compression reduces memory usage very slightly
- **Primary intersector detection** avoids failure on intersecting geometry

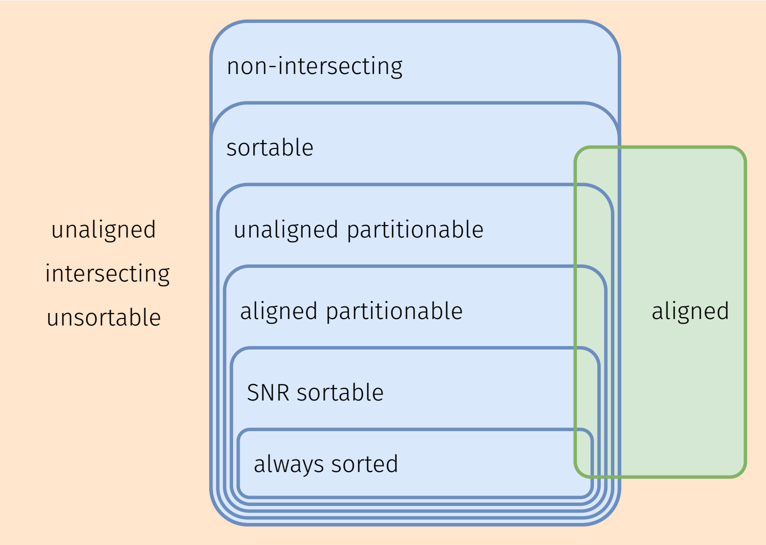
Overall Implementation Performance

- Partition tree sorting is faster than distance sorting by 60%
- Static visibility graph sorting avoids some dynamic sorting
- Detecting static special cases is important
- Distribution of sort types varies significantly with scene content

→ Distance sorting is never used in practice

Unaligned Partitioning

Overview of the Problem Space



unaligned

intersecting

unsortable

non-intersecting

sortable

unaligned partitionable

aligned partitionable

SNR sortable

always sorted

aligned

Unaligned Partitioning

Problem (Unaligned Partitioning (UAP))
Given a set of n partitionable quads Q, does a valid and useful partition R exist, and what are its parameters?

- Only known-orientation partitioning is fast

→ Large solution space of unknown-orientation partitions

- No hard cases of UAP naturally appear in Minecraft

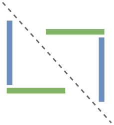


Figure 16: Aligned geometry can require an unaligned partition.

UAP in Parameter Space

- **Slope-offset parametrization** $y \geq ux + vz + d$ of planes yields parameter space $\mathbb{L} = \mathbb{R}^3$

→ **Linear mapping** from $p \in \mathbb{R}^3$ to the parameter-space half-space containing all $R \in \mathbb{L}$ for which p is in the real-space half-space $H_R^{=}$

→ Parameter-space faces correspond to real-space vertexes

- **Valid partition sets** and **useful partition sets** as expressions over parameter-space half-spaces

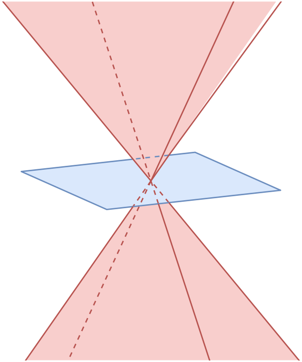
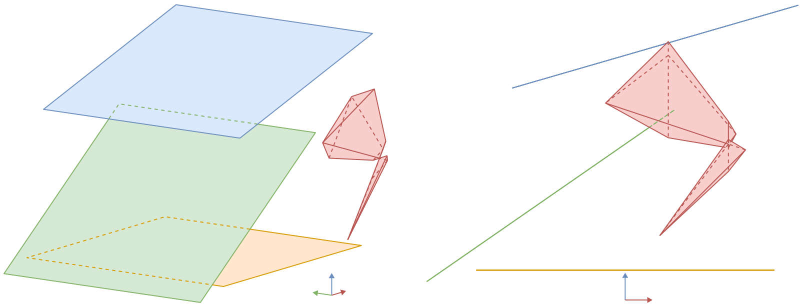


Figure 17: Valid partition set highlighted in red for blue quad q.

Useful Partitions in Parameter Space



(a) Full (b) Right side

Figure 18: Useful partition set highlighted in red for the three quads.

Upper Bounds on UAP

- **Combinatorial solution** in $\Theta(n^4)$
  1. Pick all triples of vertexes to form a plane
  2. Test the partition for validity and usefulness
  3. Mitigate a **global useless autopartition** with an artificial vertex
- Incremental solution with **linear constraint sets (LCS)** possible, but worse time complexity
- Can also be solved with LCS-SAT

Lower Bounds on UAP

- Depending on the input constraints to UAP, it can solve LCS
- **Transfer of lower bound** on LCS to UAP

→ Maps half-space constraints to vertices

- Requires more quads and expensive collision avoidance if UAP is strict

Conclusion

Conclusion

- Usually translucency requires OIT
- Minecraft's specific type of geometry makes quad-based translucency sorting tractable

→ Taking advantage of many special cases and well partitionable geometry

- Ground-truth results are possible, faster than distance sorting
- Correct translucent rendering without hardware support
- Unaligned partitioning is infeasible, but polynomial

Outlook

- Is an acceleration structure for the reduced visibility condition feasible and useful?
- Can visibility graph sorting within the partition tree improve its characteristics?
- To what extent can convex partitioning, a superset of unaligned partitioning, form a bridge between partition and graph-based sorting?

Thank you for listening!